Workstealing Schedulers for Multicore Systems
Overview

• History
• CILK and X10 dynamic parallel task models
• Work-stealing Scheduler Concepts
• Work-First v. Help-First Scheduling
• Scalable Locality-aware Work-stealing
Work-stealing: history

• Seminal work
  – Early implementations in 1981-90 for functional programs
  – Blumofe and Leicerson [1999] – CILK and WS
Work-stealing: today
Load Balancing
Distributed Systems Environment
 [Eager, Lazowska, Jahorjan, UW - 1986]

• Large-grained tasks (processes) in a NOW
  – Sender-initiated
  – Receiver-initiated
  – Issues: overhead; definition of high/low load; thrashing
  – Conclusions:
    • Sender-initiated outperforms receiver-initiated under light to moderate system loads
    • Receiver-initiated outperforms sender-initiated under heavy loads
    • Cost issue: RI often steals tasks that have already begun execution; SI can offload tasks before they begin.

• Major area of research through the 1990s (NOW, clusters) and 2000s (grids, p2p)
Work-Stealing
Shared Memory Environment

• Fine-grained tasks spawned from a common ancestor

Figure 4. Cilk’s and X10’s spawn trees (solid lines) with join edges (dashed lines) for computation generated by program in Figure 3
Languages w/ Dynamic Task Parallelism

• Cilk (MIT 1994 --) – C-like language in which user explicitly specifies the parallelism, leaving it to scheduler to distribute the threads.

```cilk
int fib (n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n - 1);
        y = spawn fib (n - 2);
        sync;
        return (x + y);
    }
}
```
CILK Work-stealing Scheduler: *Work-first*

- One worker per core
- Each worker has a deque of tasks (represented by stack frames)
- Initially, one worker executing main (the root task) and all other workers with empty deques.
- An idle worker attempts to steal tasks from other workers’ deques.
Work-first Scheduler (cont)

• On a spawn continuation is saved in a frame which is pushed onto the worker’s deque.
• Whenever worker returns from a spawned task, it checks to see if the frame it pushed before the spawn has been stolen.
• If so, the deque is empty and this worker becomes a thief.
• Else, it executes the continuation of the spawn.
**Work-first Scheduler (cont)**

- On a *spawn* continuation is saved in a frame which is pushed onto the worker’s deque.
- Whenever worker returns from a spawned task, it checks to see if the frame it pushed before the spawn has been stolen.
- If so, the deque is empty and this worker becomes a thief.
- Else, it executes the continuation of the spawn.
Work-first Scheduler (cont)

• Runtime overhead of steals increases as number of workers increases.
  – Example:
    • One busy worker creates N tasks; N-1 idle workers
    • To distribute the N tasks to the N-1 idle workers, continuations passed from victim to thief serially.

• Therefore *work-first* better when stealing is rare.
**Help-first** Scheduler

- Worker executes the continuation and leaves the spawned task to be stolen.
- (Worker asks for “help” from others before working on the new task itself.)
- More efficient because steal operations can be performed in parallel using non-blocking operations.
Threshold = Min. FIB Parameter Implemented sequentially

Figure 12. Execution times of $\text{Fib}(45)$ using 64 workers for HFP (Help-First Policy) and WFP (Work-First Policy) with thresholds 5, 10, 15, and 20 on an UltraSparc T2 system.

More parallelism $\leftarrow$----$\rightarrow$ less parallelism
Figure 15. Comparison of work-sharing, HFP (Help-First Policy) and WFP (Work-First Policy) on a 64-thread UltraSparc T2 SMP. The execution times were obtained using “Best of 30 runs” approach.
Work-first vs. Help-first

• Performance:
  – Work-first better for recursive parallelism with deep task spawn tree
  – Help-first better for flat parallelism with flat-wide spawn tree.

• Stack and Memory: ???

• Both suffer from cache contention
Scalable Locality-aware Work-stealing (SLAW)

• Adaptively selects WFP or HFP at runtime.
• Establishes bounds on stack and heap space needed.
• Locality-aware by increasing data re-use within a worker and among groups of workers.
Scalable Locality-aware Work-stealing (SLAW)

![Graph showing speedup over serial version for different benchmarks.]

Fig. 9. Performance results on Niagara 2. Deployment is locality-oblivious (1-place, 64 workers) with no processor binding.